ESD-TDR-64-636

W-07191

ESRC

PAT, A LANGUAGE FOR PROGRAMMING AND MAN-COMPUTER COMMUNICATION

TECHNICAL DOCUMENTARY REPORT NO. ESD-TDR-64-636

JUNE 1965

R. Silver
C. Wells

Prepared for

DIRECTORATE OF COMPUTERS
ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
UNITED STATES AIR FORCE
L. G. Hanscom Field, Bedford, Massachusetts

Project 508

Prepared by

THE MITRE CORPORATION
Bedford, Massachusetts
Contract AF 19(628)-2390

When US Government drawings, specifications, or other data are used for any purpose other than a definitely related government procurement operation, the government thereby incurs no responsibility nor any obligation whatsoever; and the fact that the government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data is not to be regarded by implication or otherwise, as in any manner licensing the holder or any other person or corporation, or conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

ESD-TDR-64-636 W-07191

PAT, A LANGUAGE FOR PROGRAMMING AND MAN-COMPUTER COMMUNICATION

TECHNICAL DOCUMENTARY REPORT NO. ESD-TDR-64-636

JUNE 1965

R. Silver
C. Wells

Prepared for

DIRECTORATE OF COMPUTERS
ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
UNITED STATES AIR FORCE
L. G. Hanscom Field, Bedford, Massachusetts

Project 508

Prepared by

THE MITRE CORPORATION
Bedford, Massachusetts
Contract AF 19(628)-2390

PAT, A LANGUAGE FOR PROGRAMMING AND MAN-COMPUTER COMMUNICATION

ABSTRACT

PAT is a computer language of the macro-assembly type. The program, which translates PAT into computer code, is designed to be used not only as a compiler of programs, but as a symbolic interface between a user and a computer. In this latter capacity, it can serve to interpret commands and accept command definitions for such programs as a text editor, on-line debugger, or simulated desk calculator.

The language and the translator have been designed to allow the structure of the translator itself to be modified by certain definitions encountered during the translation process.

The rules for defining symbols and referring to them have been organized to facilitate combining independently written programs into a single unit.

REVIEW AND APPROVAL

This technical documentary report has been reviewed and is approved.

SEYMOUR JEFFERY
Major, USAF
Chief, Computer Division
Directorate of Computers

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont'd.)

LIST OF TABLES

PREFACE

The translator described briefly in this report is in many respects quite similar to other contemporary assembler-compilers. For example, the format of typical assembly-level statements may be quite similar to that of most free-field assembler statements.

This language and its processor have been designed, however, to do all of the translation necessary for communication with a host of on-line programs in a time-shared system, as well as the more normal functions expected of such a program. As a result, it incorporates certain features which are perhaps different from what one might expect.

For those familiar with assemblers, the following brief discussion describes the basic features of the language and its processor.

The translator is a two-pass processor. The fundamental assembly algorithm is the well-known one in which an operation-code followed by a sequence of expressions is processed according to a property list associated with each op-code. The processing is such that each expression is evaluated in turn and subjected to a masking and shifting operation before being combined by a logical OR with the base of the op-code. Any given argument, or field expression, may be subjected to more specific manipulation if necessary.

Pseudo-operations, as such, do not exist in this system; instead their place is taken by actors which differ from pseudo-operations in that they may appear at any point within the text.

Macro-operations do not exist in this system; instead, their place is taken by macros which differ from macro-operations in that they may appear at any point within the text, and that their expansion may range from none, or one symbol to whole subprograms.

The macros, themselves, are of the type which perform a purely textual transformation on the program string and which are handled by one processor rather than by separate macro generators. The textual transformation performed is on the level of atoms, rather than on the character level, as in some systems. Both open and closed macro forms are provided, the open form appearing much like a macro-operation in use, and the closed form appearing like a function.

The function of single-word "literals" is provided as a degenerate case of a "remote code" actor which processes a subprogram into a remote location.

All definitions of the translator result from some act of an actor. Among the items which may be defined at processing time are:

(1) field properties for use in the basic assembly algorithm,

(2) operation codes,

(3) labels,

(4) macros (both open and closed),

(5) binary connectors for use in expressions evaluated at process time,

(6) actors, and

(7) internal code used to modify the translator itself.

A rather involved regional naming scheme is provided which bears some resemblance to the block scheme of Algol. The major difference between the two schemes is that each block is named, and any identifier in the text may have associated with it specific block names to force the evaluation of that particular identifier within a particular block or context. This scheme was originally intended to provide the necessary discrimination between user and system symbols when the translator was used as an on-line interpreter for system programs; it has also proved useful in other ways.

Repetitive processing of a string with sequential substitution of variables is provided by a "rep" actor which performs the function of the IRP pseudo-operation in Macro-FAP at any point within the textual string.

Conditional processing of substrings is provided by a "cs" actor, which is similar to the conditional statement of LISP. This actor works at any point within the textual string.

This translator itself is programmed as a closed subroutine which accepts a sequence of characters as input and produces an object-program file in what is essentially a "load tape" form. Since the translator is recursive, it is possible for it to call itself to process subsections of the text. This feature allows the binary-connector, actor, and internal code definition capabilities to be implemented in a straightforward manner.

Originally, the thought was to use the translator as a closed subroutine under the direction of the various system programs when on-line translation was desired. Though this is possible, it has proven advantageous to merely define appropriate actors within a copy of the translator in order to provide the necessary system programs. In doing so, the tables within the translator become a part of the system program and are subject to more intimate manipulation, the full capabilities of the translator still being available.

In addition to the bibliography at the end of this report, we have found that the literature relating to translators, especially on the assembler level, is often in the form of user manuals published by manufacturers.

SECTION I

INTRODUCTION

PAT is intended to serve as a linguistic link between man and computer in an on-line environment. The on-line computer is a tool which can be used by a man to facilitate his work. The tangible presence of the computer is a user console, conveniently located in the user's office. We will suppose this console to be a typewriter, suitably modified so that the material typed by the user is transmitted to the computer, and also so that material generated by the computer is transmitted to the typewriter and printed by it.

An on-line computer system should be responsive to the user's typed requests for computation, getting results back to him in as short a time as possible. The PAT translator is designed with a bias toward speed of compilation, at the expense of efficiency of the resulting program, in an effort to satisfy this requirement.

PAT consists of a macro-assembly language, similar to Macro-FAP. The basic alphabet of characters, and rules for constructing meaningful strings of characters are oriented toward typewriters, rather than punched cards.

The translator is a closed subroutine, containing within itself no input-output operations. It relies on the program which calls it to provide textual input on demand, and to accept output on demand, whether it be compiled code, listing text, or error diagnostics.

PASS 1

The input character string is segmented to form a sequence of symbols. This sequence is processed, allocating space for code, making definitions, and expanding macros. Pass 1 produces two things: definitions which are added to the symbol-table and intermediate output.

PASS 2

The input to pass 2 is the intermediate output of pass 1. This information is processed, producing binary output, listing, and error diagnostics, and expanding macros not encountered during pass 1.

SECTION II

SYNTAX OF THE LANGUAGE

SEGMENTATION

A program written in PAT consists of a sequence of atoms. Each atom is a string of characters in the PAT alphabet. The alphabet and the rules of segmentation by which a string of characters is broken up to form a sequence of atoms are described in the following paragraphs.

ALPHABET

The alphabet* of PAT consists of 55 characters, divided into six classes:

(1) Letters and digits (36):

a b c d e f g h i j k l m n o p q r s t u v w x y z 0 1 2 3 4 5 6 7 8 9

(2) Punctuation Characters (5):

( ) , : ;

(3) Special Characters (9):

$ / \ + - * = ← ↑

(4) Quotes (2):

\ /

(5) Function Characters (3):

space, tab, and carriage-return, which print as follows:

▯ ↦ ⤸

*We consider a subset of the available alphabet in order to ease the presentation.

ATOMS

There are two classes of atoms: identifiers, and punctuators.

A punctuator is a one-character string consisting of one punctuation character.

An identifier is one of three things:

(1) A string of characters, all of which are either letters or digits.

(2) A string of characters, the first of which is a left-quote, the last of which is a right-quote, containing no other right-quote.

(3) A one-character string consisting of one special character.

SEGMENTATION RULES

Given a string of characters σ, the segmentation rules specify how it is broken up into a sequence of atoms.

The rules given below operate on σ to produce at most one atom α, together with a residual character string σ'. The original character string is transformed into a sequence of atoms by applying the rules repeatedly to the successive residual strings until a null string is produced. The appropriate rule to apply to σ depends on the initial character ι of σ.

(1) (Alphanumeric identifier) If ι is a letter or a digit, then α is the largest initial segment of σ containing only letters or digits. α is an identifier. σ' is the string remaining when the initial segment α is deleted from σ. For brevity; we write $\sigma' = \sigma\text{-}\alpha$.

(2) (Quoted identifier) If ι is a left-quote, then the identifier α is that initial segment of σ containing just one right-quote. (The string σ is improper if there is no such segment.) $\sigma' = \sigma-\alpha$.

(3) (Punctuator) If ι is a punctuation character, then the punctuator α is ι itself, and $\sigma' = \sigma-\iota$.

(4) (⤸) If ι is "⤸", then the punctuator α is ";", and $\sigma' = \sigma-\iota$.

(5) (Special character) If ι is a special character, then the identifier α is ι itself, and $\sigma' = \sigma-\iota$.

(6) (Space or tab) If ι is a space or tab, then no atom is generated, and $\sigma' = \sigma-\iota$.

(7) (Comment) If ι is right-quote, then let β be the largest initial segment of σ not containing "⤸". No atom is generated, and $\sigma' = \sigma-\beta$.

A quoted identifier consists literally of all the characters from its initial left-quote to its terminal right-quote. Blanks and tabs are ignored outside quoted identifiers. Comments are introduced by a right-quote outside a quoted identifier, and cause all succeeding characters up to, but not including the next "⤸" to be ignored. A "⤸" is converted into the atom ";". Note that the strings "a □ b" and "ab" segment differently, whereas the strongs "a-b" and "a □ - □ b" segment the same, since "-" is not a letter or digit.

<u>Example</u>

The string "=2a↦add ˋx □ y ˊ, 2⤸24; ˋ⤸ˊˊ a;b⤸" appears as follows when typed:

```
=2a        add ˋx yˊ  , 2
24;ˋ
ˊˊ a;b
```

Segmented, it produces the following sequence of atoms:

```
=
2a
add
`x ▯ y´
,
2
;
24
;
`⤸´
;
```

These segmentation rules imply that PAT is a free-field language, the order in which the atoms occur being important, but not their placement on the typed page.

P-FORMED SEQUENCE

The sequence σ of atoms is parenthetically well-formed (p-formed) if σ contains an equal number of left-parentheses and right-parentheses, and every initial segment of σ contains at least as many left-parentheses as right-parentheses.

The parenthesis level (p-level) of an occurrence of an atom α in a p-formed sequence σ is the difference between the number of left-parentheses and the number of right-parentheses in the initial segment of σ which precedes α.

SECTION III

ASSEMBLER-LEVEL STATEMENTS

INSTRUCTIONS

An instruction consists of an op-code symbol followed by a list of arguments, each of which is an expression. If there is more than one argument field, the successive arguments are separated by commas. For example, "add a-5, 1" is an instruction.

A symbol defined as an op-code has a value consisting of a base number, plus a list of field specifications. Each field specification consists of a mask number and an offset instruction.

An instruction is processed by evaluating each argument-field expression, masking it by combining it with the appropriate mask number by a logical AND operation, then shifting it by performing the appropriate offset instruction, then combining the result with the base number by a logical OR operation.

If the instruction has fewer arguments than the op-code has fields, then the instruction is processed as if the missing fields were zero, e.g., "add a-5;" is equivalent to "add a-5, 0;".

EXPRESSIONS

An expression is an algebraic formula evaluated (by integer arithmetic) at assembly time. The binary connectors which can be used in forming expressions include "+", "-", "*", "/", and "\" (remainder). For example if a22 and b are constants with values of -3 and 14 respectively, then the expression "(2*a22+37)/10-b\ 3" has a value of +1.

PSEUDO-OPERATION

A certain set of expressions behave like conventional pseudo-operations. For example, the symbols "org" and "end" behave like their conventional counterparts. The statement "org 100;" sets the location counter to 100. The statement "end start;" specifies that the program starting address is the value of "start".

SECTION IV

SEMANTICS

This section explains what symbols are and what a context is. The semantics of PAT is determined by the contextual meanings of symbols. The symbol-table is the repository of meanings; those given initially and those created by definitions occurring during processing of a program. Thus the semantics of the PAT language is a dynamic thing, defined only with respect to a given program.

MOLECULES

The sequence of atoms constituting a program can be grouped to form a sequence of molecules. The colon is the glue which binds atoms into molecules. A molecule is either a punctuation atom other than ":", or an identifier, or a sequence of identifiers separated by colons. For example: The sequence of molecules arising from the string "a+:b:$:c;d:e" is a, +:b:$:c, ;, d:e.

TAILS

A tail is a (possibly null) sequence of identifiers. At any time during the processing of a program, there exists something called the current tail, which helps to establish the context in which symbols are defined and interpreted.

DEFINITIONS

To define is to establish the meaning of a symbol within a context by the concrete process of making an entry in the symbol-table. These entries consist of four parts: an identifier, a tail, a type, and a value. The translator makes a definition when a defining operator is encountered in the program. These operators have one property in common: The symbol defined is determined by the molecule which follows the defining operator, together with the current tail.

This molecule must be either an identifier ω or a sequence of identifiers $\omega:\omega_1:\cdots:\omega_n$. (We regard the former as a special case of thc latter in which n = 0.) Let the current tail be the sequence of identifiers $\tau_1, \cdots, \tau_m$. Then in all cases, the symbol-table entry has atom ω and tail $\omega_1, \cdots, \omega_n, \tau_1, \cdots, \tau_m$. The type of the entry is defined according to the particular operator used, and the value depends on the defining statement itself. We say that the symbol ω has been defined in the context $\omega_1, \cdots, \omega_n, \tau_1, \cdots, \tau_m$.

Examples

		Defined:	
Molecule	Current tail	Symbol	Context
abc	null	abc	null
z22	x, 4y	z22	x, 4y
z22:x	4y	z22	x, 4y

INTERPRETATION OF SYMBOLS

The translator is continually called upon to determinc the meaning of a symbol in a context. Formally, it encounters a molecule; if the molecule is a punctuation atom, then the meaning is intrinsic in the atom itself; if the molecule consists of a single atom and the atom is composed wholly of characters whose values* are less than the radix**, then the meaning is taken to be the appropriate numeric value after conversion. Otherwise the translator obtains a symbol-table entry determined by this molecule together with the current tail.

*The characters have values determined by the simple collating sequence 0, 1, 2, . . . , 9, a, b, . . . , z.

**The translator maintains a variable called the radix (nominal value ten) with respect to which all numeric conversions are made.

Let the molecule Ω be $\omega:\omega_1:\cdots:\omega_n$, and let the current tail be $\tau_1, \cdots, \tau_m$. The translator examines the symbol table to determine if there is an entry whose identifier is ω, and whose tail is $\omega_1, \cdots, \omega_n, \tau_k, \cdots, \tau_m$. If there is no such entry, then the symbol ω is not defined in this context. If there is such an entry, then the translator uses the one for which <u>k</u> is least. Thus the translator searches for a meaning for Ω in wider and wider contexts until it either finds an entry, or finds none at all.

<u>Examples</u>

		Successive entries searched for:	
<u>Molecule</u>	<u>Current tail</u>	<u>Atom</u>	<u>Tail</u>
abc	null	abc	null
z22	x, 4y	z22	x, 4y
		z22	4y
		z22	null
z22:x	4y	z22	x, 4y
		z22	x

SECTION V

SYMBOL-TABLE ENTRIES

The symbol-table entries may be classified according to type. There are seven types of entries: constant, op-code, field-specification, field set, actor, macro, and binary-connector. To each type there corresponds a value format (see Table I).

Table I

Symbol-Table Entries

Type	Value Format
Constant	A number occupying one memory word.
Field-specification	Mask, occupying one memory word. Offset, an accumulator shifting instruction.
Field-set	A list of fields, each having been previously defined by a field specification command.
Op-code	Base, a number occupying one memory word. Field set, a reference to a symbol of type fieldset; used to control the processing of fields.
Actor	Starting location of a subroutine within the translator itself.
Macro	Number of arguments. Number of created atoms. Skeleton.
Binary-connector	Starting location of a subroutine within the translator. Precedence: a number.

SECTION VI

UNIFORM HANDLING OF SYMBOLS

The basic assembly process involves translating a statement of the form "op-code $\text{field}_1, \cdots, \text{field}_n$;" to produce a word of code. To do this, the translator starts by fetching a symbol, i.e., fetching a molecule from the current source of atoms and looking it up in context, discovering in this case that the symbol is an op-code. The translator proceeds to evaluate the argument fields, which involves this same process of fetching symbols.

If the fetched symbol is an actor, then the buck is passed to the subroutine specified by the value of the actor. It is the responsibility of that routine to provide a symbol with which translation can be resumed.

If the fetched symbol is a macro, then the buck is passed to the macro expander. This routine fetches the macro argument sequences from the current source and saves them. It then obtains the macro skeleton from the symbol table entry, and changes the current source so that source atoms will be fetched from the macro skeleton (or argument sequences) until the skeleton is exhausted.

By handling symbol fetching in a uniform way, we can generalize the notion of pseudo-operation to that of actor, and the notion of macro-operation to that of macro. Any symbol fetched, be it operator-like, argument-like, or connector-like, may be an actor, forcing a subroutine call, or it may be a macro, forcing substitution of the macro skeleton into the atom stream.

SECTION VII

MACROS

MACRO FORMULAS

A macro formula is a sequence of atoms of the form "$\mu(\alpha_1, \cdots, \alpha_n)$", where μ is a symbol of type macro with n or more arguments, and the α_k are macro arguments.

A macro argument is a p-formed sequence of atoms containing no commas or semicolons at p-level zero.

MACRO EXPANSION

When the translator encounters a macro formula beginning with the symbol μ, it responds by calling the macro expander, which performs the following steps.

(1) Obtain from the entry for μ the number of a of arguments, the number c of created atoms, and the skeleton σ.

(2) Fetch the macro arguments $\alpha_1, \cdots, \alpha_n$ as sequences of atoms. If any α_k is of the form (β), where β is p-formed, then the outer parentheses are stripped off, i. e. , the sequence β, rather than (β), is used.

(3) Construct a dummy placeholder table for the a + c placeholders that may be encountered in the skeleton (see dmac, in Section VIII.)

(4) Pair each (stripped) argument sequence α_k with the placeholder $d_k(k = 1), \cdots, n)$. If $n < a$, then pair each of the placeholders $d_{n+1}, \cdots, d_a$ with the null sequence.

(5) Create c atoms, and pair them with the placeholders $d_{a+1}, \cdots, d_{a+c}$.

(6) Save the current atom source, and substitute for it a source which will extract atoms on demand from the skeleton σ.

(7) Using this new source, fetch the current symbol, and return.

The macro source has two peculiarities. First, if it obtains an atom which is the dummy placeholder d_k, then it pushes down the current source (skeleton), and switches over to a source which extracts atoms from the sequence paired with d_k. Second, when a macro source exhausts the sequence it is operating on, it restores the previously saved source. The process of saving and restoring sources is such that there are no restrictions on the type of symbol which may be encountered during translation of a macro.

CREATED ATOMS

The atoms that are created during macro expansion are unique and distinct from any atoms that the user may write. They are of the form . . 1, . . 2, etc.

Example

The definition "dmac sum a b c (fetch a; add b; store c)" having been made, the macro formula "sum(x, y, z)" will expand into "fetch x; add y; store z", while the macro formula "sum(s, y, (z, w))" will expand into "fetch s; add y; store z, w".

OPEN FORM OF MACRO FORMULA

If the user thinks of the macro μ as a macro-operation, he will prefer to write his macro formulas in the form "$\mu\ \alpha_1, \cdots, \alpha_n$", which is like an instruction, rather than "$\mu(\alpha_1, \cdots, \alpha_n)$", which is like a function. The first of these is called an open macro formula. Either type may be used, but the type must be specified when the macro symbol is defined.

SECTION VIII

ACTORS

When the translator encounters a symbol of type actor, it calls the subroutine specified by the value of the symbol-table entry. When that subroutine returns, it has established some other symbol as the current symbol. For each of the actors described below, this current symbol is the source symbol next following the last argument of the actor, unless otherwise noted.

PSEUDO-OPERATIONS

The following actors behave like pseudo-operations.

org ϵ

The actor org sets the translator's location counter to the value of the expression ϵ.

bss ϵ

The actor bss adds the value of the expression ϵ to the location counter.

end ϵ

The actor end sets the starting address of the object program to the value of the expression ϵ.

block ω

The current tail is a (possibly null) sequence of identifiers $\omega_1, \cdots, \omega_n$. It is reset by block to $\omega, \omega_1, \cdots, \omega_n$.

endblock

If the current tail is $\omega_1, \cdots, \omega_n$, with $n > 0$, then endblock resets it to $\omega_2, \cdots, \omega_n$.

ics σ

σ is a p-formed string of atoms. The actor ics generates output words containing the string of characters representing the print names of the argument atoms. Punctuation and non-quoted identifiers are represented directly, but the initial and final quotes of a quoted atom are elided. The right-quote cannot be represented directly in a string, nor can the function characters be gracefully represented. To overcome this difficulty, we let the character-pairs $r, $c, $s, $t, and $$ represent not themselves, but the characters right-quote, carriage-return, space, tab, and dollar, respectively.

re

Under normal conditions, if an actor tries to define a symbol which is already defined, an error condition will arise, and the definition will not be made. The actor re forces the next following definition to be made generating a new entry in the symbol table, and suppresses the error indication if the symbol was already defined.

change

The actor change behaves like re except that the existing symbol table entry for the next definition will be changed, rather than a new entry formed.

ACTORS WHICH DEFINE SYMBOLS

= Ω

The actor "=" defines the molecule Ω to be of the constant type with a value of that of the location counter at the moment.

equ Ω ϵ

The molecule Ω is defined by equ to be of the constant type with a value of that of the expression ϵ.

syn $\Omega\,\Omega'$

The molecule Ω is defined by syn to be of the type and value of Ω'.

dmac $\Omega\omega_k, \delta_k, \sigma$

The ω_k form a sequence of m identifiers, and the δ_k form a sequence of n identifiers. ω, δ and σ are p-formed sequences of atoms. The molecule Ω is defined by dmac to be of type macro, number of arguments m, number of created atoms n, and skeleton σ', where σ' is like σ except for having an indexed dummy placeholder substituted for each occurrence of an ω_k or a δ_k. If n = 0, then the comma may be omitted. Ω may only be used in a closed macro formula.

dmaco $\Omega\,\omega_k, \delta_k, \sigma$

Except that Ω may only be used in an open macro formula, dmaco performs exactly as dmac.

dfield $\Omega\ \epsilon, \epsilon'$

The molecule Ω is defined by dfield to be of the field-specification type with mask and offset equal to the values of the expressions ϵ and ϵ', respectively.

dfieldset $\phi\ \Omega_1\ \Omega_2 \cdots \Omega_n$

ϕ is defined by dfieldset to be the list consisting of field Ω_1, field Ω_2, ... field Ω_n.

opd $\Omega\ \phi_k, \epsilon$

The molecule Ω is defined by opd to be of type op-code, with base the value of expression ϵ, and ϕ is a fieldset. ϕ_k is a symbol of type fieldset.

dactor $\Omega \lambda$

Ω is defined by dactor to be of the actor type with value (starting location) equal to the value of the expression λ.

dbinconn $\Omega \lambda, \pi$

The molecule Ω is defined by dbinconn to be of the binary-connector type with starting location the value of the expression λ, and precedence the value of the expression π.

PSEUDO-ARGUMENTS

$

$ pretends to be a constant with value equal to the current location.

intcode (σ)

σ is a p-formed sequence. The translator is called by incode to translate the program σ in such a way that the binary code produced is placed into memory as part of the translator itself. The current symbol is then set by intcode to the constant type with the value of the starting location produced by translation. The "λ" field (starting location) in a dactor or dbinconn statement will generally be of the form intcode (σ).

I (σ)

σ is a p-formed sequence. I calls the translator to translate the program σ in such a way that all but the first word w of binary code produced is thrown away. I then sets the current symbol to the constant type with value w. The "ϵ" field (offset) in a dfield statement will usually be of the form I (σ).

loc (σ)

In pass 2, the translator maintains a list of generated words $w_1, \cdots, w_n$, and the location λ into which w_1 is to be loaded. The value w of the first word is obtained by loc which then searches the list for an occurrence of w, adding $w = w_{n+1}$ to the end of the list if it does not find it. In any case, w is then some w_k, and loc sets the current symbol to the constant type with a value of $\lambda+k$.

<u>$0(\epsilon)$</u>

0 evaluates the expression ϵ with all numerals evaluated as octal, and then sets the current symbol to the constant type with a value that of ϵ.

<u>$D(\epsilon)$</u>

D evaluates the expression ϵ with all numerals evaluated as decimal, and then sets the current symbol to the constant type with a value that of ϵ.

CONDITIONAL SEQUENCE $cs((\sigma_1)\epsilon_1, (\sigma_2)\epsilon_2, \cdots, (\sigma_n)\epsilon_n)$

The expressions ϵ_k are evaluated, one after the other, by cs. If all of them have value zero, then the whole cs-sequence is ignored. If not all of them have value zero, then the translator behaves as if it encountered the (p-formed) sequence σ_k instead of the whole cs-sequence, where ϵ_k is the first expression to have a non-zero value. For example "cs((fetch a; store b;)a-b)" is equivalent to "fetch a; store b;" if <u>a-b</u> has non-zero value (i. e. , if $a \neq b$); otherwise it is equivalent to the null sequence.

REPEAT SEQUENCE rep $\omega, \delta_k(\sigma)(\alpha_1, \cdots, \alpha_n)$

The actor rep operates as follows. A dummy macro μ is defined as if by "dmac μ $\omega, \delta_k(\sigma)$". In place of the sequence "rep. . . " the translator sees the sequence "$\mu(\alpha_1)\cdots\mu(\alpha_n)$".

The rep actor can be used within a macro skeleton in the form "rep $\omega, \delta_k(\sigma)(\alpha)$", where α is one of the arguments of the macro. Given a particular macro formula in which "$(\beta_1, \cdots, \beta_n)$" appears as the argument

α, the parentheses will be stripped off before substitution for α, and so there will be n repetitions. Of course n may vary from one macro formula to the next. For example, consider the definition "dmaco call name args (branch name; rep z(arg z;)(args))". Given this definition of the macro call, the statement "call sub, (a, (b, 5), (c, 2))" will expand into "branch sub; arg a; arg b, 5; arg c, 2;".

SECTION IX

PAT AS AN INTERPRETIVE LANGUAGE FOR ON-LINE DEBUGGING

THE DEBUGGING PROBLEM

The user who has written a program is confronted with the problem of getting that program to perform properly. Between him and this goal stand all the errors that he has made from conceptual errors in the design of the program to blunders in coding. Experience shows that the user can confidently expect some errors to evade the most careful scrutiny of his program. The only way to proceed is for him to attempt to run the program, to observe its behavior, to attempt to deduce errors from this behavior, and to fix the errors thus uncovered. The user remains in this debugging loop until the program behaves satisfactorily.

When one considers debugging practice on a conventional off-line computer, one notes that the turn-around time (the time required to go once around the debugging loop) is long, typically half a day to two days. Since the user wishes to minimize the total time needed to debug his program, he uses debugging tools which give him a maximum amount of information about how the program behaved. The principal tool is the dump: printed information produced by the computer showing the contents of selected portions of the computer memory at the time of the dump in a more or less appropriate form. In preparing for a debugging run, the user tries to anticipate all the contingencies that he can think of. The run usually produces an enormous amount of information, most of which turns out to be superfluous. He observes the case that has actually arisen, diagnoses it using the pertinent information, and ignores the rest. It is not hard to trace the difficulty to its source: the fact that the user cannot peer into the computer while his program is running and cannot stop the program at the moment a bug occurs.

ON-LINE DEBUGGING

An on-line, fast-response computer has two properties that can be used to resolve this difficulty decisively: user consoles, and fast response to requests for trivial computation. Such a system can be used to explore events as they actually occur, without ever considering contingencies which might but in fact do not arise. The user needs a debugging program tailored to on-line operation. A simplified version of such a program, showing the way in which the translator can be used to mediate communication between the user and the program follows.

The user is seated at a console, engaged in debugging a program in the computer memory. He needs to examine words in memory, change the contents of memory words, and run his program with breakpoints inserted to stop it and return control to the debugging program. He requires at the minimum a debugging program with which he can communicate, which in turn can act on the program to be debugged, and which can communicate with him. For this, he requires a language of commands which can be interpreted by the debugging program. In addition, it would be useful to have a set of symbols referring to operations and locations in his program, and the ability to define new commands. We can avoid the design of an ad hoc debugging language and translator by using PAT and the PAT translator.

A DEBUGGING PROGRAM

Suppose that we have written a program called DB, which consists of an executive routine, five subroutines, and the translator. DB itself has been written using PAT, and its symbol-table, incorporating certain symbols of DB together with the five macros listed in Table II, has been combined with that of the user.

Table II

DB Subroutines

Macro Formula	Expansion	Action of the Subroutine Called
contents a	call con:db arg a	Type out contents of memory location a.
set a, w	call set:db arg a wrd w	Replace contents of memory location a with the word w.
break a	call break:db arg a	Save a, the location into which a breakpoint is to be inserted when control is transferred to the subject program.
go a	call go:db arg a	Save the contents of the breakpoint location, insert there a jump back to DB, and jump to a.
find a, b, m, w	call find:db arg a arg b wrd m wrd w	Search locations a through b. Type out location and contents for all words which look like w through the mask m.

DB waits for the user to type in a string of characters terminated by "↵". It then calls the translator, handing it the string consisting of "org workspace:db;" followed by the string just typed, followed by the string "jump wait:db; end; ↵". Translating the string produces binary code which is loaded into the workspace by DB. When the translator returns, DB performs the code. This calls one of the subroutines, with appropriate calling sequence parameters, and then jumps back to the DB executive routine to wait for a new command to be typed in.

USING THE COMMANDS

If the user types "contents data+3 ↵", the con subroutine types out the contents of data+3 where, presumably, data is a constant referring to some location in the user's program.

Suppose the user wants to find all words in a thousand word block starting at beg which have address parts (right 16 bits) equal to ijk. The proper command is

"find beg, beg+1000, oc(177777), ijk ↵"

DEFINING NEW COMMANDS

The user is likely to use the find command discussed previously over and over again, each time examining the whole program for cells containing a suspect address. The overhead of typing 32 characters can be reduced by defining a new command: a command is just a macro, and the translator accepts macro definitions. In particular it accepts:

"dmaco fa address (find beg, beg+1000, oc(177777), address) ↵".

After having typed this definition, typing "fa ijk ↵" has the desired effect.

Let us conclude with an elaborate example. A useful technique for finding bugs is to allow the program to run a little bit at a time by specifying a breakpoint, going to the program, letting it run until the breakpoint exit is encountered, having a look, setting a new breakpoint a little further on, going back to the program at the previous breakpoint location, and so on. The user types "break b; go a ↵". When DB gets control back, the user types some finds, contents, and sets. Then he types "break c; go b ↵", and so on. Suppose that he wants to define a command bgo that will not only combine the break and the go, but will also somehow remember the old breakpoint location, so that he merely has to type "bgo c ⤸". This can be done by typing:

```
    "equ oldbreak a;
    dmaco bgo x,c (syn c oldbreak; re equ oldbreak x; break x; go c) ⤸".
```

Consider what happens when the user types "bgo c ⤸". The translator will create an atom, say ..1, define it to be synonymous with oldbreak, i.e., a constant with the value of a, redefine oldbreak as a constant with the value of b, note the breakpoint at b, and go to ..1, that is, to a.

Roland Silver
R. Silver

C B Wells
C. Wells

Bibliography

Greenwald, I. D. "A Technique for Handling Macro Instructions," Comm. of ACM, V2, N11, Nov. 1959.

McIlroy, M. Douglas, Bell Telephone Laboratories, "Macro Instruction Extensions of Compiler Languages," Comm. of ACM, V3, 1960.

Naur, Peter "Revised Report on the Algorithmic Language ALGOL 60," Comm. of ACM, V6, N1, Jan. 1963.

DOCUMENT CONTROL DATA - R&D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)*

The MITRE Corporation
Bedford, Mass.

2a. REPORT SECURITY CLASSIFICATION

Unclassified

2b. GROUP

3. REPORT TITLE

PAT, A Language for Programming and Man-Computer Communication

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*

N/A

5. AUTHOR(S) *(Last name, first name, initial)*

Silver, Roland and Wells, Codie S.

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
June, 1965	34	3

8a. CONTRACT OR GRANT NO.

AF19(628)-2390

b. PROJECT NO.

508

c.

d.

9a. ORIGINATOR'S REPORT NUMBER(S)

ESD-TDR-64-636

9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*

W-07191

10. AVAILABILITY/LIMITATION NOTICES

Qualified requestors may obtain from DDC.

DDC Release to CFSTI (formerly OTS) authorized.

11. SUPPLEMENTARY NOTES

12. SPONSORING MILITARY ACTIVITY

Directorate of Computers
Electronic Systems Division
L. G. Hanscom Field, Bedford

13. ABSTRACT

PAT is a computer language of the macro-assembly type. The program, which translates PAT into computer code, is designed to be used not only as a compiler of programs, but as a symbolic interface between a user and a computer. In this latter capacity, it can serve to interpret commands and accept command definitions for such programs as a text editor, on-line debugger, or simulated desk calculator.

The language and the translator have been designed to allow the structure of the translator itself to be modified by certain definitions encountered during the translation process

The rules for defining symbols and referring to them have been organized to facilitate combining independently written programs into a single unit

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Programming Languages						
PAT						
Macro-assembly type						
Computer Languages						
PAT						

INSTRUCTIONS

1. ORIGINATING ACTIVITY: Enter the name and address of the contractor, subcontractor, grantee, Department of Defense activity or other organization *(corporate author)* issuing the report.

2*a*. REPORT SECURITY CLASSIFICATION: Enter the overall security classification of the report. Indicate whether "Restricted Data" is included. Marking is to be in accordance with appropriate security regulations.

2*b*. GROUP: Automatic downgrading is specified in DoD Directive 5200.10 and Armed Forces Industrial Manual. Enter the group number. Also, when applicable, show that optional markings have been used for Group 3 and Group 4 as authorized.

3. REPORT TITLE: Enter the complete report title in all capital letters. Titles in all cases should be unclassified. If a meaningful title cannot be selected without classification, show title classification in all capitals in parenthesis immediately following the title.

4. DESCRIPTIVE NOTES: If appropriate, enter the type of report, e.g., interim, progress, summary, annual, or final. Give the inclusive dates when a specific reporting period is covered.

5. AUTHOR(S): Enter the name(s) of author(s) as shown on or in the report. Enter last name, first name, middle initial. If military, show rank and branch of service. The name of the principal author is an absolute minimum requirement.

6. REPORT DATE: Enter the date of the report as day, month, year; or month, year. If more than one date appears on the report, use date of publication.

7*a*. TOTAL NUMBER OF PAGES: The total page count should follow normal pagination procedures, i.e., enter the number of pages containing information.

7*b*. NUMBER OF REFERENCES: Enter the total number of references cited in the report.

8*a*. CONTRACT OR GRANT NUMBER: If appropriate, enter the applicable number of the contract or grant under which the report was written.

8*b*, 8*c*, & 8*d*. PROJECT NUMBER: Enter the appropriate military department identification, such as project number, subproject number, system numbers, task number, etc.

9*a*. ORIGINATOR'S REPORT NUMBER(S): Enter the official report number by which the document will be identified and controlled by the originating activity. This number must be unique to this report.

9*b*. OTHER REPORT NUMBER(S): If the report has been assigned any other report numbers *(either by the originator or by the sponsor)*, also enter this number(s).

10. AVAILABILITY/LIMITATION NOTICES: Enter any limitations on further dissemination of the report, other than those imposed by security classification, using standard statements such as:

(1) "Qualified requesters may obtain copies of this report from DDC."

(2) "Foreign announcement and dissemination of this report by DDC is not authorized."

(3) "U. S. Government agencies may obtain copies of this report directly from DDC. Other qualified DDC users shall request through

_______________________________."

(4) "U. S. military agencies may obtain copies of this report directly from DDC. Other qualified users shall request through

_______________________________."

(5) "All distribution of this report is controlled. Qualified DDC users shall request through

_______________________________."

If the report has been furnished to the Office of Technical Services, Department of Commerce, for sale to the public, indicate this fact and enter the price, if known.

11. SUPPLEMENTARY NOTES: Use for additional explanatory notes.

12. SPONSORING MILITARY ACTIVITY: Enter the name of the departmental project office or laboratory sponsoring *(paying for)* the research and development. Include address.

13. ABSTRACT: Enter an abstract giving a brief and factual summary of the document indicative of the report, even though it may also appear elsewhere in the body of the technical report. If additional space is required, a continuation sheet shall be attached.

It is highly desirable that the abstract of classified reports be unclassified. Each paragraph of the abstract shall end with an indication of the military security classification of the information in the paragraph, represented as *(TS)*, *(S)*, *(C)*, or *(U)*.

There is no limitation on the length of the abstract. However, the suggested length is from 150 to 225 words.

14. KEY WORDS: Key words are technically meaningful terms or short phrases that characterize a report and may be used as index entries for cataloging the report. Key words must be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location, may be used as key words but will be followed by an indication of technical context. The assignment of links, rules, and weights is optional